Hindawi
Security and Communication Networks
Volume 2018, Article ID 4983404, 18 pages
https://doi.org/10.1155/2018/4983404

Research Article

A Vendor-Neutral Unified Core for Cryptographic Operations in GF(p) and GF(2^m) Based on Montgomery Arithmetic

Martin Schramm,[1,2] **Reiner Dojen**,[1] **and Michael Heigl**[2]

[1]*Department of Electronic and Computer Engineering, University of Limerick, Limerick, Ireland*
[2]*Institute ProtectIT, Deggendorf Institute of Technology, 94469 Deggendorf, Germany*

Correspondence should be addressed to Martin Schramm; martin.schramm@th-deg.de

Received 6 October 2017; Revised 14 March 2018; Accepted 17 May 2018; Published 21 June 2018

Academic Editor: Fawad Ahmed

In the emerging IoT ecosystem in which the internetworking will reach a totally new dimension the crucial role of efficient security solutions for embedded devices will be without controversy. Typically IoT-enabled devices are equipped with integrated circuits, such as ASICs or FPGAs to achieve highly specific tasks. Such devices must have cryptographic layers implemented and must be able to access cryptographic functions for encrypting/decrypting and signing/verifying data using various algorithms and generate true random numbers, random primes, and cryptographic keys. In the context of a limited amount of resources that typical IoT devices will exhibit, due to energy efficiency requirements, efficient hardware structures in terms of time, area, and power consumption must be deployed. In this paper, we describe a scalable word-based multivendor-capable cryptographic core, being able to perform arithmetic operations in prime and binary extension finite fields based on Montgomery Arithmetic. The functional range comprises the calculation of modular additions and subtractions, the determination of the Montgomery Parameters, and the execution of Montgomery Multiplications and Montgomery Exponentiations. A prototype implementation of the adaptable arithmetic core is detailed. Furthermore, the decomposition of cryptographic algorithms to be used together with the proposed core is stated and a performance analysis is given.

1. Introduction

The next generation of embedded systems and IoT devices will exhibit a much higher degree of internetworking which gives rise to security considerations [1]. As a logical consequence, such devices must become cryptographic nodes, besides others, being capable of encrypting/decrypting and signing/verifying data as well as establishing spontaneous secured communications by exchanging common secrets used for secret key calculation. While many embedded chips already have support for hardware-accelerated symmetric algorithms (mainly AES) [2] and hash functions, due to various reasons, such as complexity, space, and costs, they lack in hardware support especially for supporting a wide range of public-key and key exchange algorithms with different precision widths. Besides, many modern cryptographic primitives necessitate the capability for producing true random numbers and random prime numbers. Typical IoT devices furthermore very often only exhibit a limited amount of resources which requires efficient cryptographic hardware structures in terms of area, power consumption, and calculation performance [3]. In general enterprises developing IoT products basically have three options to include application functionalities in high integrated devices, using Application Specific Standard Products (ASSP), Application Specific Integrated Circuits (ASIC), or Field Programmable Gate Arrays (FPGA). Today FPGAs have become promising components for IoT applications [4], compared to ASSP solutions which often cannot provide the required functionality and can provide a better Total Cost of Ownership (TCO) compared to ASIC solutions. Thus for devices which are equipped with a FPGA device, it is valuable to examine how efficient hardware structures for performing cryptographic operations can be included.

In matters of algorithm agility an arithmetic engine with minimal hardware footprint, which can handle the arithmetic

operations of a great variety of cryptographic algorithms, is of great importance for IoT based devices. Especially the calculability of the individual operations leading to lower and upper calculation time bounds is quite important.

This paper proposes a tiny-held vendor-neutral cryptographic arithmetic core exemplarily implemented in FPGA-logic. For efficiency, time-intensive modular operations, such as multiplication and exponentiation operations, Montgomery Arithmetic is used. Without the need of any expensive software precalculations the core is able to perform a high number of cryptographic algorithms and handle various key sizes by simply processing operation lists. Furthermore the core architecture is unified and can perform calculations in both prime finite fields ($GF(p)$) and binary extension fields ($GF(2^m)$). To illustrate the versatility of the developed core, well-established cryptographic algorithms have been rewritten and fragmented into operation lists to be processed by the arithmetic engine.

The paper is organized as follows. Section 2 states the related work of this research. In Section 3 the design of the proposed Enhanced Montgomery Multiplication Core is stated; the specified functional range of the core is given in Section 4. In Section 5 some exemplary application descriptions for the core are mentioned and in Section 6 the results of the performance analysis are stated. Finally, Section 7 concludes the paper.

2. Related Works

The efficiency of cryptographic algorithms when implemented on reconfigurable hardware is mainly determined by the fact of how the underlying finite field arithmetic operations are realized [5]. Several applications in cryptography such as ciphering and deciphering of asymmetric algorithms, the creation and verification of digital signatures, and secure key exchange mechanisms require excessive use of the basic finite field modular arithmetic operations addition, multiplication, and the calculation of the multiplicative inverse. Especially the field multiplication operation is crucial to the efficiency of a design, since it is the core operation of many cryptographic algorithms [6].

In [7] P. L. Montgomery introduced a representation of residue classes in order to speed up modular multiplications without affecting modular additions and subtractions. Over the years numerous designs have been proposed implementing modular multiplications based on Montgomery's multiplication algorithm [8]. The foundation for these architectures was presented by A. Tenca and Ç. Koç in [9]. The architecture is based on a word-based Montgomery Multiplication algorithm for prime finite fields in which multiplications are performed in a bit-serial fashion. E. Savaş et al. in [10] have proposed an extension which, in addition to the standard integer modulo arithmetic, also allows polynomial computations over binary finite fields. An overview about algorithms and hardware architectures for Montgomery Multiplication can be found in [11]. Optimizations of the original design have been proposed concerning the hardware implementation of the Montgomery Multiplication algorithm [12] as well as by utilizing special arithmetic hardcore extensions of FPGAs to accelerate digital signal processing applications [13]. Some designs only focus on utilizing the Montgomery Multiplication method to accelerate modular exponentiation operations as required by the RSA algorithm [14, 15].

However, no publication focuses on how the Montgomery Multiplication architecture can be embedded into a comprehensive solution. In this paper we propose an enhanced version of a bit-serial word-based unified Montgomery Multiplication core based on logic elements only which is controlled by a state machine and offers the functional range to be able to perform complete cryptographic algorithms without additional complex processing required in software.

3. Enhanced Montgomery Multiplication Core

3.1. Requirements. Today a high number of different public-key algorithms are in use. To ensure compatibility, cryptographic applications must support a large portion of those algorithms. While typical software implementations often can easily be upgraded in order to adapt new algorithms and larger key sizes, the same is not necessarily true for hardware implementations. Therefore following requirements have been identified for the Enhanced Montgomery Multiplication Core:

(i) *Use of Montgomery Arithmetic*. The design must be able to perform modulo operations in a time-efficient manner by using Montgomery Arithmetic. At least the core must support Montgomery Multiplications and Montgomery Exponentiations. Furthermore the core must support standard modulo additions and modulo subtractions.

(ii) *Works on Both Finite Fields $GF(p)$ and $GF(2^m)$*. The architecture must exhibit an unified structure supporting both standard integer modulo operations of prime finite fields as well as polynomial calculations of binary finite fields.

(iii) *Montgomery Parameter Calculation*. In general the Montgomery Parameters (r and r^2) can be precomputed for previously known moduli. However, as a requirement the core must be able to handle arbitrary moduli. Therefore it must be capable of calculating the Montgomery Parameters $r \bmod n$, $r^2 \bmod n$ and $r^{-1} \bmod n$ without the need of precalculations done in software.

(iv) *Scalable Design*. The architecture must be scalable in terms of timing, area, and power consumption. This includes the parametrisation of the word width, the internal storage size, and the amount of processing units within the pipeline.

(v) *Multialgorithm Support*. The core must be based on a building-block design. The functional range provided by the arithmetic unit should empower algorithm agility, by fragmenting cryptographic algorithms into a list of core operations. At least the core must be capable of performing RSA [16] operations, (safe) prime number generation and primality testing (MR)

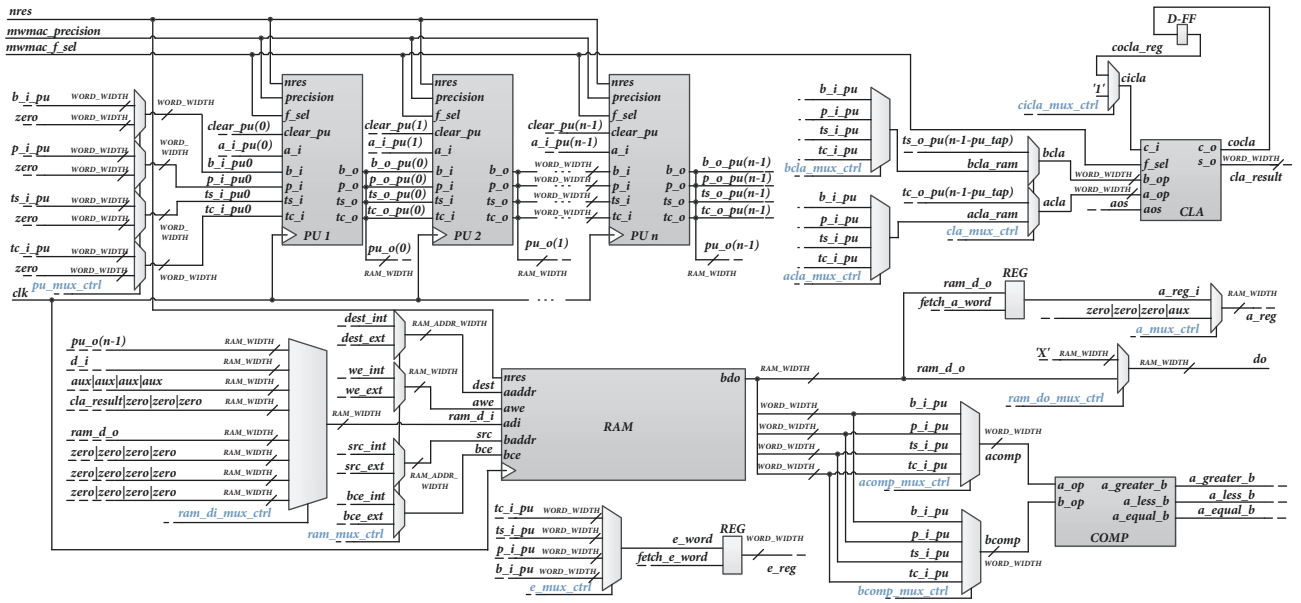

Figure 1: Overall architecture of the Enhanced Montgomery Multiplication Core.

[17, 18], key exchange operations (DH) [19], and elliptic curve calculations (EC) [20] over both prime and binary finite fields.

(vi) *Supporting as Many Precision Widths as Possible*. The design must support a wide range of different precision widths determining the security level of the cryptographic algorithm. If a certain security level, due to increased attacking computing power, becomes inadequate, the precision width can be adjusted accordingly which makes the hardware less prone to become obsolete due to higher security demands. The core must support the current recommendations for minimum key sizes [21] and should also support larger key sizes. For RSA algorithm and Diffie-Hellman key exchange support the architecture should be able to handle precisions up to 4096 bit moduli, for elliptic curve cryptography support precisions up to 512 bits for prime finite fields and precisions up to 571 bits for binary finite fields should be possible.

(vii) *Time-Invariant Operations*. The architecture must be capable of performing its operations in a time-invariant manner. If security sensitive information, such as private keys, will be processed, it must be ensured that all operations exhibit the same execution time to prevent side-channel attacks based on timing analysis.

3.2. Overall Core Architecture. Figure 1 illustrates the overall architecture of the proposed Enhanced Montgomery Multiplication Core which is capable of meeting all requirements as specified above.

Besides the pipeline of processing units handling the main part of the word-based Montgomery Multiplication algorithm, the core features an enhanced word-based Carry Look-Ahead adder being responsible for the calculation of the final result after the pipeline has processed all bits of an operand as well as for performing single modular addition and subtraction operations. The register files of the original design have been replaced with an internal dual-ported RAM which holds the operands as well as intermediate results of the core operations. Furthermore a word-based comparator component has been described which is queried during operations to decide if a modular addition or subtraction step must be performed. Two additional r-bit words for the A operand and the exponent E have been introduced with $|r|$ being the RAM width ($|r| = 4 \cdot |w|$) which will be fetched from RAM in case of Montgomery Multiplication and Montgomery Exponentiation operations. An auxiliary w-bit word *aux* is used for RAM reorganisation operations as well as for the calculation of the Montgomery Parameters r and r^2.

The intelligence of the core is the controlling state machine which utilizes the defined components to perform standard modular addition and subtraction operations, Montgomery Multiplications, Montgomery Exponentiations, Montgomery Parameter calculation, and RAM reorganisation operations. Therefore it is responsible for controlling the RAM write and read access, the source and destination address signals of RAM, as well as the values passed through to the first processing unit, to the CLA adder, and to the comparator component. Furthermore it controls the assignments of A operand, E exponent, and *aux* words.

The described core can be parametrised in three ways. The parameter named *MAX_PRECISION_WIDTH* specifies the highest supported precision width $|p|$, whereas the parameter *WORD_WIDTH* is used to specify the word width $|w|$ of the operands involved in the calculations. These two parameters determine the size and the address space of the internal core RAM. The third parameter *MAX_NUM_PUS* specifies

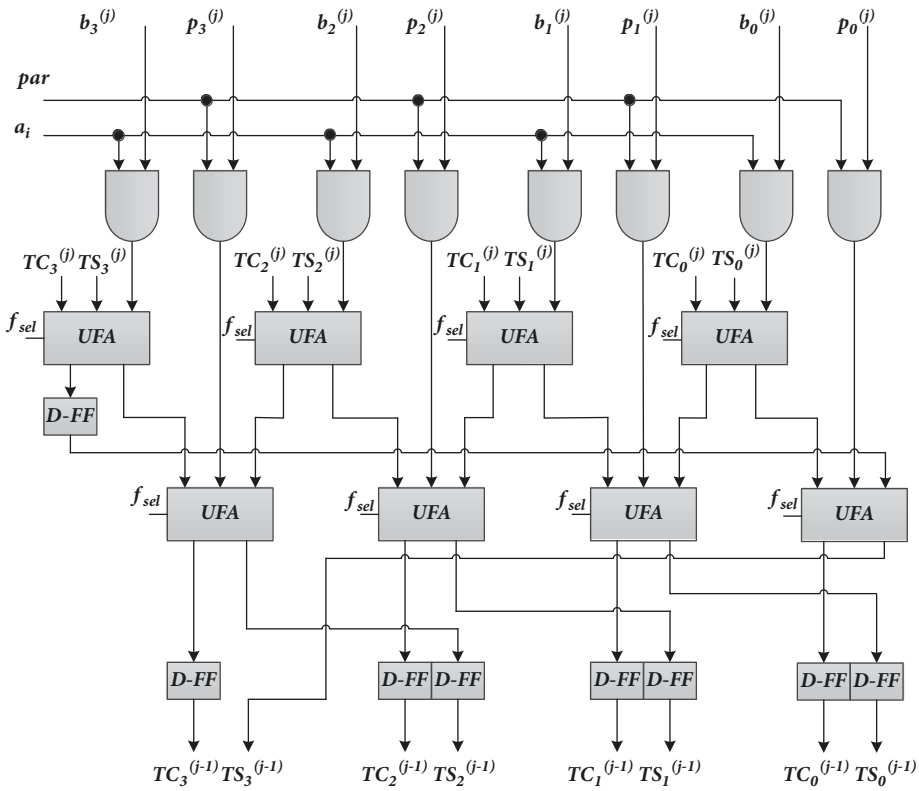

Figure 2: Processing unit with word size $w = |4|$.

the maximum number of processing units of the pipeline implemented for a specific core variation mainly affecting the performance and the size in terms of area consumption.

3.2.1. Processing Units. The heart of the core is the pipeline of processing units implementing the multiple word version of the Montgomery Multiplication algorithm. Therefore the processing unit structure has been described from scratch. The processing unit can be held in reset and keeps track of the cycle number according to the number of words to be processed depending on the supplied parameters. This control logic is needed to determine whether the supplied modulus has to be added to the processed words in this cycle or not, depending on the value of the signal *par* denoting an odd intermediate result. Note that buffering the output of a processing unit between two processing units is not required in this design. Compared to the original design presented in [10] for a given precision width $|p|$ and a word size $|w|$, $e = \lceil |p|/|w| \rceil + 1$ number of words are required for a unified solution and the pipeline must consist of a power of two (2^x) number of processing units with a maximum number of $2^x < (e-1)$ in order to avoid pipeline stalls. Figure 2 illustrates the internal architecture of an exemplary processing unit with word size $|w| = 4$.

Each processing unit consists of a cascade of two layers of so-called Unified Full Adder (UFA) cells. The Unified Full Adder cells basically consist of simple full adder cells which have been enhanced by an additional finite field selection input f_{sel}. This allows for the creation of a unified multiplier architecture which can not only be used in prime fields $GF(p)$ ($f_{sel} = 1$) but also in binary fields $GF(2^m)$ ($f_{sel} = 0$) in which additions will be simple bitwise XOR calculations without any carry output.

3.2.2. Carry Look-Ahead Adder. Since the pipeline generates the result in carry save form, an additional step is necessary at the end of each calculation to obtain a nonredundant version of the result. For the sake of uniformity a circuit is required that can operate in both finite fields $GF(p)$ and $GF(2^m)$. Furthermore, since the calculation in $GF(p)$ could require one further subtraction step, the Carry Look-Ahead adder in the design has been formulated to be able to perform word-based modular additions and subtractions. Figure 3 illustrates the logic of the proposed enhanced n–bit wide CLA adder of the core.

The internal signal b' of the second operand will be calculated as $b_i' = b_i \oplus (aos \cdot f_{sel})$ in which *aos* denotes an add-or-subtract signal ($aos = 0$ means addition, $aos = 1$ represents subtraction by performing an addition in two's complement representation). The modified CLA adder involves the same common Carry Look-Ahead adder logic for the calculation of the generate ($g_i = a_i \cdot b_i'$) and propagate ($p_i = a_i + b_i'$) functions. The output values c_i of the CLA adder logic will be calculated as $c_0 = c_{in}$ for the least-significant bit and $c_i = g_{i-1} + (p_{i-1} \cdot c_{i-1})$ for all further bits. The final sum output bits s_i will be calculated as $s_i = (c_i \cdot f_{sel}) \oplus a_i \oplus b_i'$ the carry output bit will be determined as $c_{out} = c_n \cdot f_{sel}$. If the selected finite field is $GF(2^m)$ ($f_{sel} = 0$), then the add-or-subtract input will

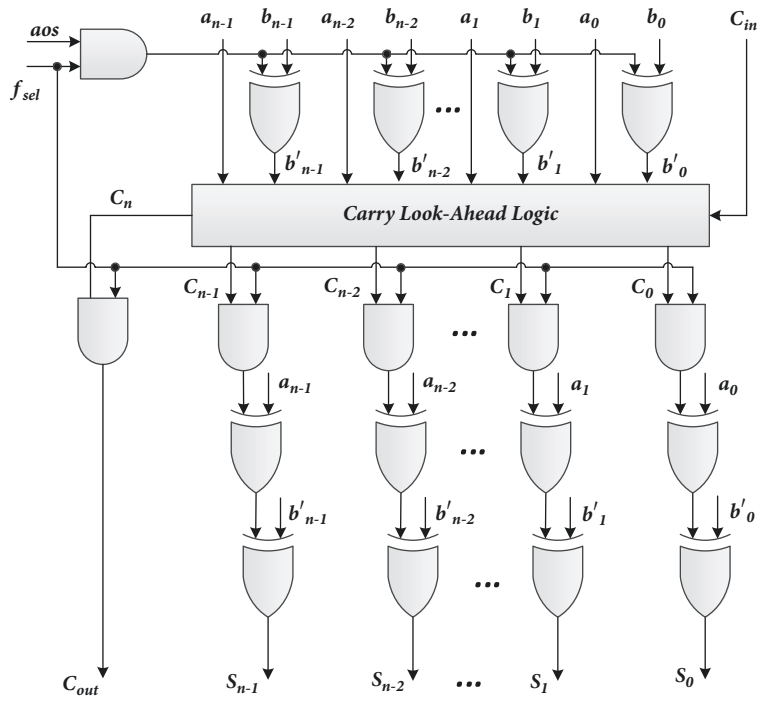

Figure 3: Enhanced n-bit wide CLA adder.

X_8	X_7	X_6	X_5	X_4	X_3	X_2	X_1	E_8	E_7	E_6	E_5	E_4	E_3	E_2	E_1	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	$B_{8\ 7}$	$B_{7\ 6}$	$B_{6\ 5}$	$B_{5\ 4}$	$B_{4\ 3}$	$B_{3\ 2}$	$B_{2\ 1}$	$B_{1\ 0}$
																								$P_{8\ 15}$	$P_{7\ 14}$	$P_{6\ 13}$	$P_{5\ 12}$	$P_{4\ 11}$	$P_{3\ 10}$	$P_{2\ 9}$	$P_{1\ 8}$
																								$TS_{8\ 23}$	$TS_{7\ 22}$	$TS_{6\ 21}$	$TS_{5\ 20}$	$TS_{4\ 19}$	$TS_{3\ 18}$	$TS_{2\ 17}$	$TS_{1\ 16}$
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	$TC_{8\ 31}$	$TC_{7\ 30}$	$TC_{6\ 29}$	$TC_{5\ 28}$	$TC_{4\ 27}$	$TC_{3\ 26}$	$TC_{2\ 25}$	$TC_{1\ 24}$

Figure 4: Enhanced Montgomery Multiplication Core RAM organization.

be ignored, the final sum will simply be the bitwise modulo-2 addition of the two input values *TC* and *TS* and the carry output bit will be forced to zero.

3.2.3. Core RAM Structure. The RAM of the core must be capable of holding all the necessary operands and intermediate values required during the execution of cryptographic algorithms. The basic structure of the described RAM is pictured in Figure 4.

It features four symbolic horizontal RAM operand locations with *MAX_PRECISION_WIDTH* bit each which are organized as eight pieces of *MAX_PRECISION_WIDTH*/8 bit each. The location named *B* is intended to hold operand *B* in Montgomery Multiplication and Montgomery Exponentiation operations; the location named *P* is intended to hold the modulus. The location *TS* usually holds the temporary sum value during Montgomery Multiplications and Montgomery Exponentiation or the first operand in modular addition or subtraction operations. The location *TC* usually holds the temporary carry stream during Montgomery Multiplications and Montgomery Exponentiation or the second operand in modular addition or subtraction operations.

Besides the horizontal RAM operand locations three symbolic vertical RAM operand locations with *MAX_PRECISION_WIDTH* bit each have been defined which are organized as eight pieces of *MAX_PRECISION_WIDTH*/8 bit each. The locations named *A*, *E*, and *X* for convenience usually are used to hold operand *A* in Montgomery Multiplication and Montgomery Exponentiation operations as well as the exponent operand *E* and the auxiliary operand *X* in Montgomery Exponentiation operations. In addition all RAM slots are intended to hold intermediate values during the execution of cryptographic algorithms.

4. Functional Range of the Core

This section provides a description of the functional range of the proposed core. The following precisions (denoted in bit-length) are supported:

(i) *EC over GF(p), RSA, MR, DH*: 192, 224, 256, 320, 384, 448, 512, 768, 1024, 1536, 2048, 3072, 4096

(ii) *EC over GF*(2^m): 131, 163, 176, 191, 193, 208, 233, 239, 272, 283, 304, 359, 368, 409, 431, 571

If further or other precision widths should be supported, the described core can easily be adjusted in an appropriate manner. For the parametrisation and the execution/abortion of an operation a 32-bit wide command input word has been defined. Besides the start, abort, and finite field selection signals also the encoded precision width, operation code as well as RAM offsets for the specified operation can be supplied. The following operations have been specified.

4.1. MontMult Operation. The *MontMult* operation code instructs the core to perform a single Montgomery Multiplication with the supplied elements in the given finite field. A Montgomery Multiplication will start by reading the first r-bit word of operand A from RAM. Afterwards the pipeline will be started and the appropriate bits of A operand will be fed to the individual processing unit. If all bits of the A operand word have been fed to the processing units, a new word will be read from RAM. Once the last bit of A operand has been processed, the temporary sum and temporary carry words will be fed into the CLA adder in order to reunite the two streams. After the last words of temporary sum and temporary carry have been brought together, the carry output bit of the CLA adder will be evaluated. If a carry bit is set the modulus will be subtracted once; otherwise the result will be compared to the given modulus. If the result is equal or greater than the modulus the given modulus will be subtracted once.

4.2. MontR Operation. The *MontR* operation code instructs the core to calculate the Montgomery Parameter $r = 2^k$ regarding a supplied modulus in the given finite field, with k being the bit-length of the given precision.

In the case of prime field arithmetic the Montgomery Parameter r will be $r \equiv 2^k \bmod p$, so r can be calculated as two's complement of p as bitwise inverse of the given modulus plus 1. Therefore the individual words of the modulus will be XOR-ed with a constant word consisting of all-ones. In addition the least-significant bit of the first word will be set to one.

In the case of binary field arithmetic the Montgomery Parameter r will be $r \equiv 2^k \bmod n(x)$, so r is equal to binary expression of the irreducible polynomial $n(x)$ with the most significant bit set to zero. Therefore the individual words of the modulus will be scanned and the appropriate most significant bit will be set to zero, depending on the given precision.

4.3. MontR2 Operation. The *MontR2* operation code instructs the core to calculate the Montgomery Parameter r^2 with $r^2 = 2^{2 \cdot k}$ for a supplied modulus in the given finite field with k being the bit-length of the given precision.

In the case of prime field arithmetic the Montgomery Parameter r^2 will be given by $r^2 \equiv r \cdot r \equiv 2^k \cdot 2^k \equiv 2^{2 \cdot k} \bmod p$. Therefore in a first step the Montgomery Parameter $r \equiv 2^k \bmod p$ will be calculated for prime fields as described above. In order to calculate r^2 one possible way is to calculate $2^l \cdot 2^k \bmod p$ with l being a small divider of k. In the given implementation $l = 1$. Therefore the bits of r will be shifted to the left by one bit. If the result is equal or greater than the modulus, p will be subtracted once. By using a square-and-multiply-like algorithm, multiple Montgomery Multiplications will be performed in order to calculate $r^2 \equiv 2^k \cdot 2^k \bmod p$.

In the case of binary field arithmetic the Montgomery Parameter r^2 will be given by $r^2 \equiv r \cdot r \equiv 2^k \cdot 2^k \equiv 2^{2 \cdot k} \bmod n(x)$. Therefore in a first step the Montgomery Parameter $r \equiv 2^k \bmod n(x)$ will be calculated for binary fields as described above. In order to calculate r^2 the resulting parameter r will be shifted k-times bitwise to the left. After each shift, the most significant bit as given by the precision parameter will be evaluated. If the bit is one, the irreducible polynomial will be added to the intermediate result which represents a modulo reduction with $n(x)$. Once the shift has been performed k-times the result will be $r^2 \equiv 2^k \cdot 2^k \bmod n(x)$

4.4. MontExp Operation. The *MontExp* operation code instructs the core to perform a Montgomery Exponentiation consisting of multiple Montgomery Multiplication steps in the given finite field. A Montgomery Exponentiation will start by reading the first r-bit word of exponent E from RAM. Afterwards the first appearing one of the exponent word will be searched starting from the most significant bit. If the first word consists of all-zeros then the next word of exponent E will be read and evaluated. Once the highest bit of exponent E has been found, multiple Montgomery Multiplications will be performed until all bits of the exponent have been processed following a square-and-multiply algorithm.

4.5. ModAdd Operation. The *ModAdd* operation code instructs the core to perform a modular addition of the supplied elements in the given finite field. After preparing the core for the addition operation, the CLA adder will add the given operands using the appropriate arithmetic given by the finite field selection input. Once the last words of the given operands have been added the carry output bit of the CLA adder will be evaluated. If a carry bit is set, the modulus will be subtracted once; otherwise the result will be compared to the given modulus. If the result is equal to or greater than the modulus, it will also be subtracted once.

4.6. ModSub Operation. The *ModSub* operation code instructs the core to perform a modular subtraction of the supplied elements in prime fields. After preparing the core for the subtraction operation the CLA adder will be used to perform a word-based subtraction by performing an addition in two's complement representation with prime field arithmetic. After the last words of the given operands have been processed, the carry output bit of the CLA adder will be evaluated. If the carry bit signals a negative result, the modulus will be added once; otherwise the result will be compared to the given modulus. If the result is equal to or greater than the modulus, it will be subtracted once.

4.7. RAM Copy Operations. In order to support cryptographic algorithms which have been disassembled into a list

of instructions, RAM copy operations are needed. According to the proposed RAM layout stated above four individual copy operations have been defined.

The *CopyH2V* operation code instructs the core to copy a number of words, according to the supplied precision parameter, from the horizontal RAM layout starting from the given source address to the vertical RAM layout starting from the given destination address.

The *CopyV2V* operation code instructs the core to copy a number of words, according to the supplied precision parameter, from the vertical RAM layout starting from the given source address to the vertical RAM layout starting from the given destination address.

The *CopyH2H* operation code instructs the core to copy a number of words, according to the supplied precision parameter, from the horizontal RAM layout starting from the given source address to the horizontal RAM layout starting from the given destination address.

The *CopyV2H* operation code instructs the core to copy a number of words, according to the supplied precision parameter, from the vertical RAM layout starting from the given source address to the horizontal RAM layout starting from the given destination address.

4.8. MontMult1 Operation. The *MontMult1* operation code instructs the core to perform a single Montgomery Multiplication of the supplied element with the constant 1 in the given finite field. This type of operation is needed when a montgomerized value should be transformed back from the Montgomery Domain and has been implemented as an independent operation since an operand $A = 1$ will unnecessarily occupy a vertical RAM slot. A Montgomery Multiplication with the constant 1 will be executed in an analogous manner as the *MontMult* operation with the only exception that, instead of the RAM words, constant words will be used for the A operand.

5. Exemplary Core Application Descriptions

This section gives exemplary descriptions of how the specified functional range of the proposed building-block Enhanced Montgomery Multiplication Core design can be utilized to support a wide range of cryptographic algorithms demanding the least possible memory capacity yet at the same time supporting as much precision widths as possible. Information is given of how to perform Chinese Remainder Theorem [22] (CRT) accelerated RSA private key operations and how to use the core in order to test/generate prime numbers.

For the support of elliptic curve cryptography over prime and binary finite fields modular functions are given for preparing and conducting point operations for arbitrary elliptic curves for the supported precision widths. For all these algorithms a list of operations and the quantity of different operations is given allowing to perform cryptographic algorithms by simply processing these operation lists.

5.1. CRT-Accelerated RSA Operation. In order to speed up RSA private key operations the CRT-accelerated version is also supported by the core. Therefore some operations have to be performed with full precision whereas most of the operations have to be performed with half precision. Algorithm 1 lists the necessary steps to utilize the core for CRT-accelerated RSA private key operations.

Requires: $(c, d, p, q, exp1, exp2, coeff, n)$
Calculates: (m)

(h) $rq^2 = MontR2(q)$;
(f) $cq = MontMult(rq^2, c, q)$;
(h) $cq_{MD} = MontMult(rq^2, cq, q)$;
(h) $mq_{MD} = MontExp(cq_{MD}, cq_{MD}, exp2, cq_{MD}, q)$;
(h) $mq = MontMult1(mq_{MD}, q)$;
(h) $rp^2 = MontR2(p)$;
(f) $cp = MontMult(rp^2, c, p)$;
(h) $cp_{MD} = MontMult(rp^2, cp, p)$;
(h) $mp_{MD} = MontExp(cp_{MD}, cp_{MD}, exp1, cp_{MD}, p)$;
(h) $mp = MontMult1(mp_{MD}, p)$;
(h) $coeff_{MD} = MontMult(rp^2, cp, p)$;
(h) $t1 = ModSub(mp, mq, p)$;
(h) $t2 = MontMult(coeff_{MD}, t1, p)$;
(f) $r^2 = MontR2(n)$;
(f) $q_{MD} = MontMult(q, r^2, n)$;
(f) $t3 = MontMult(t2, q_{MD}, n)$;
(f) $m = ModAdd(t3, mq, n)$;
Provides: (m)

Algorithm 1: CRT-RSA private key operation.

Table 1 illustrates the abstract operations lists of the core for CRT-accelerated RSA application using the private key portion for all supported precision widths (512, 768, 1024, 1536, 2048, 3072, 4096). The number given in the index of the RAM locations denotes the offset given by the corresponding src_addr, dest_addr, src_addr_e, src_addr_x input signals. The width of the processed values depends on the supplied mwmac_precision input signal which depends on the operation. In the table operations requiring full precision (the precision of the RSA modulus) are marked by (f), operations requiring half precision are marked by (h). The mwmac_f_sel signal must be set to $GF(p)$ arithmetic.

CRT-accelerated RSA private key operations require $2 \times MontR2$, $4 \times MontMult$, $2 \times MontMult1$, $2 \times MontExp$, $1 \times ModSub$, $9 \times CopyH2V$, $4 \times CopyV2H$, $2 \times CopyV2V$ and $1 \times CopyH2H$ performed on half precision and $1 \times MontR2$, $4 \times MontMult$, $1 \times ModAdd$, $1 \times CopyH2V$ and $1 \times CopyH2H$ performed on full precision.

5.2. Prime Generation/Testing Operation. Algorithm 2 lists the necessary steps to utilize the core, in conjunction with a TRNG generator as Miller-Rabin Primality Tester. In the algorithm n denotes the random integer to be tested for primality and k denotes the confidence parameter determining the accuracy of the test, i.e., the amount of Miller-Rabin loops. In a precomputation step the parameters s and d with $2^s \cdot d = (n - 1)$ must be calculated which can be done by simple shift operations and counter increments in software.

Table 1: Core operations list CRT-RSA.

Step Nr.	Precision	Operation
1	-	Clear RAM
2	-	Write $q \mapsto P_1$, $exp1 \mapsto E_1$, $exp2 \mapsto E_5$, $coeff \mapsto X_1$, $p \mapsto X_5$
3	(h)	$MontR2(P_1, A_1)$
4	(h)	$CopyH2V(B_1, A_1)$
5	-	Write $c \mapsto B_1$
6	(f)	$MontMult(A_1, B_1, P_1)$
7	(h)	$MontMult(A_1, B_1, P_1)$
8-9	(h)	$CopyH2V(B_1, A_1)$, $CopyV2V(A_1, A_5)$
10	(h)	$MontExp(A_5, B_1, E_5, A_1, P_1)$
11	(h)	$MontMult1(B_1, P_1)$
12 - 14	(h)	$CopyH2V(P_1, E_5)$, $CopyV2H(X_5, P_1)$, $CopyH2V(B_1, X_5)$
15	(h)	$MontR2(P_1, A_1)$
16	(f)	$CopyH2V(B_1, A_1)$
17	(h)	$CopyV2H(X_1, B_1)$
18	(h)	$MontMult(A_1, B_1, P_1)$
19	(h)	$CopyH2V(B_1, X_1)$
20	-	Write $c \mapsto B_1$
21	(f)	$MontMult(A_1, B_1, P_1)$
22	(h)	$MontMult(A_1, B_1, P_1)$
23 - 24	(h)	$CopyH2V(B_1, A_1)$, $CopyV2V(A_1, A_5)$
25	(h)	$MontExp(A_5, B_1, E_1, A_1, P_1)$
26	(h)	$MontMult1(B_1, P_1)$
27 - 28	(h)	$CopyH2H(B_1, TS_1)$, $CopyV2H(X_5, TC_1)$
29	(h)	$ModSub(TS_1, TC_1, P_1)$
30	(h)	$MontMult(X_1, B_1, P_1)$
31 - 32	(h)	$CopyH2V(B_1, A_1)$, $CopyH2V(TS_1, X_1)$
33	-	Write $n \mapsto P_1$
34	(f)	$MontR2(P_1, A_5)$
35	(h)	$CopyV2V(X_1, A_5)$
36	(f)	$MontMult(E_5, B_1, P_1)$
37	(f)	$MontMult(A_1, B_1, P_1)$
38	(f)	$CopyH2H(B_1, TS_1)$
39	(h)	$CopyV2H(X_5, TC_1)$
40	(f)	$ModAdd(TS_1, TC_1, P_1)$

The test furthermore requires an amount of random integers $\{a_1, \ldots, a_k\}$ serving as random bases.

Table 2 illustrates the operations list of utilizing the core for Miller-Rabin Primality Test steps for all supported precision widths (192, 224, 256, 320, 384, 512, 768, 1024, 1536, 2048, 3072, 4096). The number given in the index of the

Precomputation: $(s, d \text{ with } 2^s \cdot d = (n-1))$
Input: $(n, s, d, k\ \{a_1, \ldots, a_k\})$
Output: (*eval*, composite or probably prime)

$r = MontR(n)$;
$(n - r) = ModSub(n, r, n)$;
for i ***from*** 1 ***to*** k **do**
  $t1_{MD} = MontExp(a_i, a_i, d, a_i, n)$;
  $t2_{MD} = ModSub(t1_{MD}, r, n)$;
  $t3_{MD} = ModSub(t1_{MD}, (n - r), n)$;
  if $t2_{MD} = 0$ ***or*** $t3_{MD} = 0$ **then**
    continue;
  for j ***from*** 0 ***to*** $(s - 1)$ **do**
    $t1_{MD} = MontMult(t1_{MD}, t1_{MD}, n)$;
    $t2_{MD} = ModSub(t1_{MD}, r, n)$;
    if $t2_{MD} = 0$ **then**
      return (*eval* = composite);
    $t3_{MD} = ModSub(t1_{MD}, (n - r), n)$;
    if $t3_{MD} = 0$ **then**
      continue;
  return (*eval* = composite);
return (*eval* = probably prime)

Algorithm 2: Modified Miller-Rabin Primality Test.

RAM locations denotes the offset given by the corresponding src_addr, dest_addr, src_addr_e, src_addr_x input signals. The width of the processed values depends on the supplied mwmac_precision input signal. The mwmac_f_sel signal must be set to $GF(p)$ arithmetic. Note that since the results of the performed operations will be in the Montgomery Domain, they will be checked against the Montgomery Parameter r and $(n - r)$ instead of 1 and $(n - 1)$. Also note that the random bases a_i that will be checked must not necessarily be transformed into the Montgomery Domain first, they simply will be interpreted as random montgomerized values.

The total number of needed core operations depends on the security parameter k and the value s resulting from the factorization of $(n-1)$. Within the outer for loop from writing a new a_i to the RAM until the evaluation of $t2_{MD}$ $1 \times MontR$, $1 \times MontExp$, $1 \times ModSub$, $1 \times CopyH2V$, $2 \times CopyV2H$, $1 \times CopyV2V$ and $1 \times CopyH2H$ and until evaluation of $t3_{MD}$ $2 \times ModSub$, $2 \times CopyV2H$ and $2 \times CopyH2H$ operations are required. Within the inner for loop until evaluation of updated $t2_{MD}$ $1 \times MontMult$, $1 \times ModSub$, $1 \times CopyH2V$, $2 \times CopyV2H$ and $1 \times CopyH2H$ and until evaluation of updated $t3_{MD}$ $2 \times ModSub$, $2 \times CopyV2H$ and $2 \times CopyH2H$ operations is required.

5.3. Elliptic Curve Operations. Unlike modular exponentiation which only is based on modular multiplications, elliptic curve Point Addition and Point Doubling operations also in the Jacobian projective coordinate representation [23] involve modular additions, subtractions, and multiplications. The algorithms for prime field elliptic curve Point Addition and Point Doubling using Jacobian coordinates furthermore involve multiplications by some constants. Since the described core performs multiplication operations by

Table 2: Core operations list for Miller-Rabin Primality Test.

Step Nr.	Operation
1	Clear RAM
2	Write $n \mapsto P_1$, $d \mapsto E_1$
3	Write $a \mapsto X_1$
4-5	$CopyV2V(X_1, A_1)$, $CopyV2H(X_1, B_1)$
6	$MontExp(X_1, B_1, E_1, A_1, P_1)$
7	$CopyH2V(B_1, X_1)$
8	$MontR(P_1, B_1)$
9-11	$CopyH2V(B_1, A_1)$, $CopyV2H(X_1, TS_1)$, $CopyH2H(B_1, TC_1)$
12	$ModSub(TS_1, TC_1, P_1)$
13	Read B_1, if $t2_{MD} = 0$ continue at Step Nr. 3 else continue at Step Nr. 14
14-15	$CopyH2H(P_1, TS_1)$, $CopyV2H(A_1, TC_1)$
16	$ModSub(TS_1, TC_1, P_1)$
17-18	$CopyV2H(X_1, TS_1)$, $CopyH2H(B_1, TC_1)$
19	$ModSub(TS_1, TC_1, P_1)$
20	Read B_1, if $t3_{MD} = 0$ continue at Step Nr. 3 else if $(s-1) = 0$ stop test with $eval$ = composite else continue at Step Nr. 21
21	$CopyV2H(X_1, B_1)$
22	$MontMult(X_1, B_1, P_1)$
23-25	$CopyH2V(B_1, X_1)$, $CopyH2H(B_1, TS_1)$, $CopyV2H(A_1, TC_1)$
26	$ModSub(TS_1, TC_1, P_1)$
27	Read B_1, if $t2_{MD} = 0$ stop test with $eval$ = composite else continue at Step Nr. 28
28-29	$CopyH2H(P_1, TS_1)$, $CopyV2H(A_1, TC_1)$
30	$ModSub(TS_1, TC_1, P_1)$
31-32	$CopyV2H(X_1, TS_1)$, $CopyH2H(B_1, TC_1)$
33	$ModSub(TS_1, TC_1, P_1)$
34	Read B_1, if $t3_{MD} = 0$ and $i = k$ stop test with $eval$ = probably prime else if $t3_{MD} = 0$ and $i \neq k$ continue at Step Nr. 3 else if $j = (s-1)$ stop test with $eval$ = composite else continue at Step Nr. 21

using Montgomery Arithmetic, these constants must be transformed into the Montgomery Domain first for the intermediate values to remain montgomerized.

In order to utilize the core for elliptic curve operations the following modular functions have been specified for both $GF(p)$ and $GF(2^m)$ support:

(i) *EC Preparation*.

(ii) *EC Montgomery Transformation*.

(iii) *EC Affine-to-Jacobi Transformation*.

Requires: $(2, 3, 4, 8, a, b, p)$
Calculates: $(r^2, 2_{MD}, 3_{MD}, 4_{MD}, 8_{MD}, a_{MD}, b_{MD}, exp)$

$r^2 = MontR2(p)$;
$2_{MD} = MontMult(r^2, 2, p)$;
$3_{MD} = MontMult(r^2, 3, p)$;
$4_{MD} = MontMult(r^2, 4, p)$;
$8_{MD} = MontMult(r^2, 8, p)$;
$a_{MD} = MontMult(r^2, a, p)$;
$b_{MD} = MontMult(r^2, b, p)$;
$exp = ModSub(p, 2, p)$;
Provides: $(r^2, 2_{MD}, 3_{MD}, 4_{MD}, 8_{MD}, a_{MD}, b_{MD}, exp)$

Algorithm 3: Core $GF(p)$ EC Preparation.

Requires: (x_P, y_P, r^2)
Calculates: (x_{PMD}, y_{PMD})

$x_{PMD} = MontMult(x_P, r^2, p)$;
$y_{PMD} = MontMult(y_P, r^2, p)$;
Provides: (x_{PMD}, y_{PMD})

Algorithm 4: Core $GF(p)$ EC Montgomery Transformation.

(iv) *EC Point Validation*.

(v) *EC Point Doubling*.

(vi) *EC Point Addition*.

(vii) *EC Jacobi-to-Affine Transformation*.

(viii) *EC Montgomery Backtransformation*.

In the following, algorithms for utilizing the core to perform EC operations in $GF(p)$ are stated. For $GF(2^m)$ EC support, similar algorithms have been derived.

5.3.1. GF(p) EC Preparation. The prime field EC Preparation steps include the calculation of the Montgomery Parameter $r^2 \bmod p$, the exponent $\exp = p - 2$ as well as the montgomerized versions of the constants $2, 3, 4, 8$ and the EC Domain Parameters a and b for a given elliptic curve $E : y^2 \equiv x^3 + a \cdot x + b \bmod p$ over $GF(p)$. Algorithm 3 lists the necessary steps to utilize the core for EC prime field preparation.

A core prime field EC preparation operation requires $1 \times MontR2$, $6 \times MontMult$, $1 \times ModSub$, $8 \times CopyH2V$, and $8 \times CopyH2H$.

5.3.2. GF(p) EC Montgomery Transformation. The prime field EC Montgomery Transformation steps are responsible for the transformation of the supplied affine point coordinates x_P and y_P of a Point P in the case of a Point Doubling or Point Multiplication operation, x_P, y_P, x_Q and y_Q of the curve Points P and Q in the case of a Point Addition operation into the Montgomery Domain. Algorithm 4 lists

Requires: (x_{PMD}, y_{PMD})
Calculates: $(x_{PMDj}, y_{PMDj}, z_{PMDj})$

$x_{PMDj} \coloneqq x_{PMD}$;
$y_{PMDj} \coloneqq y_{PMD}$;
$z_{PMDj} = MontR(p)$;
Provides: $(x_{PMDj}, y_{PMDj}, z_{PMDj})$

Algorithm 5: Core $GF(p)$ EC Affine-to-Jacobi Transformation.

Requires: $(x_{PMD}, y_{PMD}, p, a_{MD}, b_{MD})$
Output: (*eval*, point on curve or point not on curve)

$y^2_{PMD} = MontMult(y_{PMD}, y_{PMD}, p)$;
$x^2_{PMD} = MontMult(x_{PMD}, x_{PMD}, p)$;
$ax_{PMD} = MontMult(a_{MD}, x_{PMD}, p)$;
$x^3_{PMD} = MontMult(x^2_{PMD}, x_{PMD}, p)$;
$t1_{MD} = ModAdd(x^3_{PMD}, ax_{PMD}, p)$;
$t2_{MD} = ModAdd(t1_{MD}, b_{MD}, p)$;
$t3_{MD} = ModSub(y^2_{PMD}, t2_{MD}, p)$;
if $t3_{MD} = 0$ **then**
    return: (*eval* = point on curve);
else
    return: (*eval* = point not on curve);

Algorithm 6: Core $GF(p)$ EC Point Validation.

the steps to utilize the core for prime field EC Montgomery Transformation for an arbitrary curve Point P.

A core prime field EC Montgomery Transformation operation requires $2 \times MontMult$, $2 \times CopyH2V$, and $2 \times CopyV2V$ in the case of an intended Point Doubling or Point Multiplication operation and $4 \times MontMult$ and $4 \times CopyH2V$ in the case of an intended Point Addition operation.

5.3.3. GF(p) EC Affine-to-Jacobi Transformation. The prime field EC Affine-to-Jacobi Transformation steps are responsible for transforming the supplied montgomerized affine point coordinates x_{PMD} and y_{PMD} of a curve Point P into Jacobian coordinates. Algorithm 5 lists the necessary steps to utilize the core for prime field EC Affine-to-Jacobi Transformation for an arbitrary montgomerized curve Point P.

A core prime field EC Affine-to-Jacobi Transformation operation requires $1 \times MontR$, $1 \times CopyH2V$, and $1 \times CopyV2V$ in the case of an intended Point Addition, Point Doubling, or Point Multiplication operation.

5.3.4. GF(p) EC Point Validation. The prime field EC Point Validation performs a check, if a supplied (or calculated) point indeed is a valid point of the elliptic curve given by the equation $y^2 \equiv x^3 + a \cdot x + b \bmod p$. As a requirement the Point Validation must be conducted on montgomerized points in affine coordinate representation. Algorithm 6 lists the necessary steps to utilize the core for prime field EC Point Validation.

A core prime field EC Point Validation operation requires $4 \times MontMult$, $2 \times ModAdd$, $1 \times ModSub$, $4 \times CopyV2H$, and $5 \times CopyH2H$.

5.3.5. GF(p) EC Point Doubling. The prime field EC Point Doubling steps perform a single Point Doubling operation of a Point P with montgomerized Jacobi coordinates, resulting in $2 \cdot P = R$ also represented in montgomerized Jacobi coordinates. The original algorithm for Point Doubling with Jacobi coordinate representation has been modified to be suitable for the proposed core and is given in Algorithm 7.

A core prime field EC Point Doubling operation requires $15 \times MontMult$, $1 \times ModAdd$, $3 \times ModSub$, $6 \times CopyH2V$, $6 \times CopyV2H$, and $7 \times CopyH2H$.

5.3.6. GF(p) EC Point Addition. The prime field EC Point Addition steps perform a single Point Addition operation of two Points P and Q with montgomerized Jacobi coordinates, resulting in $P + Q = R$ also represented in montgomerized Jacobi coordinates. The original algorithm for Point Addition with Jacobi coordinate representation has been modified to be suitable for the proposed core and is given in Algorithm 8.

A core prime field EC Point Addition operation requires $17 \times MontMult$, $6 \times ModSub$, $9 \times CopyH2V$, $7 \times CopyV2H$, and $12 \times CopyH2H$.

5.3.7. GF(p) EC Jacobi-to-Affine Transformation. The prime field EC Jacobi-to-Affine Transformation steps are responsible for the transformation of the supplied montgomerized Jacobi coordinates x_{RMDj}, y_{RMDj}, and z_{RMDj} of the curve Point R back into affine coordinate representation. This transformation step requires the calculation of a modular multiplicative inverse element which will be performed by a Montgomery modular exponentiation according to Euler's theorem since the modulus is a prime number. Algorithm 9 lists the necessary steps to utilize the core for prime field EC Jacobi-to-Affine Transformation.

A core prime field EC Jacobi-to-Affine Transformation operation requires $4 \times MontMult$, $1 \times MontExp$, $3 \times CopyH2V$, $1 \times CopyV2H$, $1 \times CopyV2V$ and $1 \times CopyH2H$.

5.3.8. GF(p) EC Montgomery Backtransformation. The prime field EC Montgomery Backtransformation steps are responsible for the transformation of the supplied montgomerized point coordinates x_{RMD} and y_{RMD} of a Point R out of the Montgomery Domain. Algorithm 10 lists the necessary steps to utilize the core for prime field EC Montgomery Backtransformation for an arbitrary curve Point R.

A core prime field EC Montgomery Backtransformation operation requires $2 \times MontMult1$ and $2 \times CopyV2H$.

6. Performance Analysis

In this section parameter-dependent formulas for the calculation of the computation times in clock cycles of the described basic core operations are given which allows specifying upper and lower calculation boundaries. Furthermore for the supported precision widths in both finite fields the number of words to be processed and the possible numbers

Requires: $(x_{PMDj}, y_{PMDj}, z_{PMDj}, p, 2_{MD}, 3_{MD}, 4_{MD}, 8_{MD}, a_{MD})$
Calculates: $(x_{RMDj}, y_{RMDj}, z_{RMDj})$

$t1_{MD} = MontMult(4_{MD}, x_{PMDj}, p);$
$y^2_{PMDj} = MontMult(y_{PMDj}, y_{PMDj}, p);$
$S_{MD} = MontMult(t1_{MD}, y^2_{PMDj}, p);$
$x^2_{PMDj} = MontMult(x_{PMDj}, x_{PMDj}, p);$
$t2_{MD} = MontMult(3_{MD}, x^2_{PMDj}, p);$
$z^2_{PMDj} = MontMult(z_{PMDj}, z_{PMDj}, p);$
$z^4_{PMDj} = MontMult(z^2_{PMDj}, z^2_{PMDj}, p);$
$t3_{MD} = MontMult(a_{MD}, z^4_{PMDj}, p);$
$M_{MD} = ModAdd(t2_{MD}, t3_{MD}, p);$
$M^2_{MD} = MontMult(M_{MD}, M_{MD}, p);$
$t4_{MD} = MontMult(2_{MD}, S_{MD}, p);$
$x_{RMDj} = ModSub(M^2_{MD}, t4_{MD}, p);$
$t5_{MD} = ModSub(S_{MD}, x_{RMDj}, p);$
$t6_{MD} = MontMult(M_{MD}, t5_{MD}, p);$
$y^4_{PMDj} = MontMult(y^2_{PMDj}, y^2_{PMDj}, p);$
$t7_{MD} = MontMult(8_{MD}, y^4_{PMDj}, p);$
$y_{RMDj} = ModSub(t6_{MD}, t7_{MD}, p);$
$t8_{MD} = MontMult(y_{PMDj}, z_{PMDj}, p);$
$z_{RMDj} = MontMult(2_{MD}, t8_{MD}, p);$
Provides: $(x_{RMDj}, y_{RMDj}, z_{RMDj})$

Algorithm 7: Core $GF(p)$ EC Point Doubling.

Table 3: Core RAM copy operations computation time in clock cycles (CC).

	GF(p)	**GF(2^m)**
$CC_{CopyH2V}$	$\lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 3$	$\lceil(m/\lvert w\rvert)\rceil + 3$
$CC_{CopyV2V}$	$\lceil(\lvert p\rvert/\lvert r\rvert)\rceil + 2$	$\lceil(m/\lvert r\rvert)\rceil + 2$
$CC_{CopyH2H}$	$\lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 3$	$\lceil(m/\lvert w\rvert)\rceil + 3$
$CC_{CopyV2H}$	$\lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 3$	$\lceil(m/\lvert w\rvert)\rceil + 3$

of processing units is given. In order to estimate the size ratio of different core variations the number of logic elements and dedicated logic registers for exemplary Altera and Xilinx FPGAs is stated. Furthermore results of power estimation are given. Depending on the resulting clock cycle times of core variations a reference implementation exhibiting a balance of performance and area consumption has been defined. For this reference implementation the computation times in clock cycles for the described exemplary cryptographic algorithms are given.

6.1. Core Computation Time Formulas. Table 3 lists the RAM copy operations computation time formulas in clock cycles of the proposed core. Note that the resulting calculation times of RAM reorganisation operations are only dependent on the specified precision ($|p|$ for $GF(p)$ and m for $GF(2^m)$), the word width $|w|$ parameter for which the core variation has been generated and the resulting RAM width parameter $|r|$ with $|r| = 4 \cdot |w|$. The operations $CopyH2V$, $CopyH2H$, and $CopyV2H$ exhibit the same computation time, whereas the operation $CopyV2V$ will be performed in less clock cycles.

Table 4: Core $GF(p)$ operations computation time in clock cycles (CC).

	GF(p)
$CC_{MontMult\_b_{GF(p)}}$	$\frac{\lvert p\rvert - k}{k} \cdot e + k + e + 4$
$CC_{MontMult\_w_{GF(p)}}$	$\frac{\lvert p\rvert - k}{k} \cdot e + k + 3 \cdot e + 4$
$CC_{MontR_{GF(p)}}$	$\lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 3$
$CC_{MontR2\_b_{GF(p)}}$	$2 \cdot (\lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 2) + 1 + x \cdot (CC_{CopyH2V} - 1) + y \cdot (CC_{MontMult\_b_{GF(p)}} - 1) + 1$
$CC_{MontR2\_w_{GF(p)}}$	$2 \cdot (\lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 2) + 2 \cdot \lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 3 + x \cdot (CC_{CopyH2V} - 1) + y \cdot (CC_{MontMult\_w_{GF(p)}} - 1) + 1$
$CC_{MontExp\_b_{GF(p)}}$	$[\lceil(\lvert p\rvert/\lvert w\rvert)\rceil \cdot (\lvert w\rvert + 2)] + (CC_{CopyV2V} - 1) + [2 \cdot (CC_{MontMult\_b_{GF(p)}} - 1)]$
$CC_{MontExp\_w_{GF(p)}}$	$[\lceil(\lvert p\rvert/\lvert w\rvert)\rceil] + [\lceil(\lvert p\rvert/\lvert w\rvert)\rceil \cdot \lvert w\rvert - \lvert p\rvert + 3] + [2 \cdot (\lvert p\rvert - 2) \cdot (CC_{MontMult\_w_{GF(p)}} - 1)] + [(\lvert p\rvert - 2) \cdot (CC_{CopyV2V} - 1)] + [(\lvert p\rvert - 3) \cdot (CC_{CopyH2V} - 1)] + 1$
$CC_{ModAdd\_b_{GF(p)}}$	$\lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 4$
$CC_{ModAdd\_w_{GF(p)}}$	$3 \cdot \lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 6$
$CC_{ModSub\_b_{GF(p)}}$	$\lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 4$
$CC_{ModSub\_w_{GF(p)}}$	$2 \cdot \lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 4$
$CC_{ModSub\_aw_{GF(p)}}$	$3 \cdot \lceil(\lvert p\rvert/\lvert w\rvert)\rceil + 6$

The computing time formulas of prime field core operations given in clock cycles are listed in Table 4.

Requires: $(x_{PMDj}, y_{PMDj}, z_{PMDj}, x_{QMDj}, y_{QMDj}, z_{QMDj}, p, 2_{MD})$
Calculates: $(x_{RMDj}, y_{RMDj}, z_{RMDj})$

$z^2_{QMDj} = MontMult(z_{QMDj}, z_{QMDj}, p);$
$U1_{MD} = MontMult(x_{PMDj}, z^2_{QMDj}, p);$
$z^2_{PMDj} = MontMult(z_{PMDj}, z_{PMDj}, p);$
$U2_{MD} = MontMult(x_{QMDj}, z^2_{PMDj}, p);$
$z^3_{QMDj} = MontMult(z^2_{QMDj}, z_{QMDj}, p);$
$S1_{MD} = MontMult(y_{PMDj}, z^3_{QMDj}, p);$
$z^3_{PMDj} = MontMult(z^2_{PMDj}, z_{PMDj}, p);$
$S2_{MD} = MontMult(y_{QMDj}, z^3_{PMDj}, p);$
$H_{MD} = ModSub(U2_{MD}, U1_{MD}, p);$
$R_{MD} = ModSub(S2_{MD}, S1_{MD}, p);$
if $H_{MD} = 0$ **then**
  if $R_{MD} \neq 0$ **then**
    Notify: $(x_{RMDj}, y_{RMDj}, z_{RMDj}) = (0, 1, 0)$
      Point_at_Infinity ($\mathcal{O}$);
  else
    Perform: PointDoubling Operation;
else
  $R^2_{MD} = MontMult(R_{MD}, R_{MD}, p);$
  $H^2_{MD} = MontMult(H_{MD}, H_{MD}, p);$
  $H^3_{MD} = MontMult(H^2_{MD}, H_{MD}, p);$
  $t1_{MD} = MontMult(U1_{MD}, H^2_{MD}, p);$
  $t2_{MD} = MontMult(2_{MD}, t1_{MD}, p);$
  $t3_{MD} = ModSub(R^2_{MD}, H^3_{MD}, p);$
  $x_{RMDj} = ModSub(t3_{MD}, t2_{MD}, p);$
  $t4_{MD} = MontMult(S1_{MD}, H^3_{MD}, p);$
  $t5_{MD} = ModSub(t1_{MD}, x_{RMDj}, p);$
  $t6_{MD} = MontMult(R_{MD}, t5_{MD}, p);$
  $y_{RMDj} = ModSub(t6_{MD}, t4_{MD}, p);$
  $t7_{MD} = MontMult(z_{PMDj}, H_{MD}, p);$
  $z_{RMDj} = MontMult(z_{QMDj}, t7_{MD}, p);$
  Provides: $(x_{RMDj}, y_{RMDj}, z_{RMDj})$

Algorithm 8: Core $GF(p)$ EC Point Addition.

Requires: $(x_{RMDj}, y_{RMDj}, z_{RMDj}, p, exp = p - 2)$
Calculates: (x_{RMD}, y_{RMD})

$t1_{MD} = MontExp(z_{RMDj}, z_{RMDj}, exp, z_{RMDj}, p);$
$t2_{MD} = MontMult(t1_{MD}, t1_{MD}, p);$
$t3_{MD} = MontMult(t2_{MD}, t1_{MD}, p);$
$x_{RMD} = MontMult(x_{RMDj}, t2_{MD}, p);$
$y_{RMD} = MontMult(y_{RMDj}, t3_{MD}, p);$
Provides: (x_{RMD}, y_{RMD})

Algorithm 9: Core $GF(p)$ EC Jacobi-to-Affine Transformation.

The computation time of the *MontMult* operation in $GF(p)$ depends on the specified precision $|p|$, the number of active processing units k as well as the number of words $e = \lceil (|p|/|w|) \rceil + 1$ running through the pipeline. In order to specify lower and upper computation times a best case and worst case formula is given. In the best case the carry-out bit of the CLA adder after reuniting TS and TC words is not set, the comparator only has to evaluate the most significant word, and a modular subtraction is not necessary. In the worst case the carry-out bit of the CLA adder is also not set but the comparator has to evaluate all words and a reduction of the resulting value is necessary.

The *MontR* operation in $GF(p)$ only depends on the chosen precision $|p|$ and specified word width $|w|$ parameters.

Requires: (x_{RMD}, y_{RMD})
Calculates: (x_R, y_R)

$x_R = MontMult1(x_{RMD}, p)$;
$y_R = MontMult1(y_{RMD}, p)$;
Provides: (x_R, y_R)

Algorithm 10: Core $GF(p)$ EC Montgomery Backtransformation.

Table 5: Precision-dependent values of x and y for $GF(p)$ $MontR2$ operation.

	192	224	256	320	384	448	512
x	7	6	8	7	8	7	9
y	8	11	8	10	9	12	9
	768	**1024**	**1536**	**2048**	**3072**	**4096**	
x	9	10	10	11	11	12	
y	10	10	11	11	12	12	

For the $MontR2$ operation computation time a best case and worst case formula is given. In the best case, after the shift operation, the comparator will only evaluate one word and an initial modular subtraction operation is not necessary. For the involved Montgomery Multiplication operations the best case formula is used. In the worst case, after the shift operation the comparator has to evaluate all words and decide that an initial modular subtraction operation is needed. For the involved Montgomery Multiplication operations the worst case formula is used. The amount x of $CopyH2V$ and y of $MontMult$ operations depends on the chosen precision. Table 5 lists the values for all supported $GF(p)$ precisions.

For the $MontExp$ operation, computation time in a best case and worst case formula is given. In the best case the exponent operand is 3; therefore only two Montgomery Multiplications and one $CopyV2V$ operation is necessary. For the involved Montgomery Multiplication operations the best case formula is used. In the worst case the exponent is assumed to be $2^{(|p|-1)}$; therefore $2\cdot(|p|-2)\times MontMult$, $(|p|-2)\times CopyV2V$ and $(|p|-3)\times CopyH2V$ operations have to be performed. For the involved Montgomery Multiplication operations the worst case formula is used.

For the $ModAdd$ operation computation time a best case and worst case formula is given. In the best case, after the modular addition the CLA adder carry-out bit will not be set, the comparator will only have to evaluate one word and an additional modular subtraction is not needed. In the worst case the CLA adder carry-out bit will also not be set, but the comparator will have to evaluate all words to decide that an additional modular subtraction is necessary.

For the $ModSub$ operation computation time a best case, worst case, and absolute worst case formula is given. In the best case, after the modular subtraction the CLA adder carry-out bit will not be set, the comparator will only have to evaluate one word and an additional modular subtraction is not needed. In the worst case, after the modular subtraction the CLA adder carry-out bit will be set and a modular addition must be performed. In the absolute worst case after the modular subtraction the CLA adder carry-out bit will not be set, the comparator will evaluate all words, and an additional modular subtraction step is necessary. Note that this will only occur if the resulting value after the first subtraction operation will be identical to the modulus, which under normal operation conditions will not be the case.

Table 6: Core $GF(2^m)$ operations computation time in clock cycles (CC).

Operation	$GF(2^m)$
$CC_{MontMult_{GF(2^m)}}$	$\frac{m-(m \bmod k)}{k}\cdot e + (m \bmod k) + e + 4$
$CC_{MontR_{GF(2^m)}}$	$\lceil (m/\lvert w\rvert)\rceil + 3$
$CC_{MontR2\_b_{GF(2^m)}}$	$(m+1)\cdot(\lceil (m/\lvert w\rvert)\rceil+2)+m+1$
$CC_{MontR2\_w_{GF(2^m)}}$	$(m+1)\cdot(\lceil (m/\lvert w\rvert)\rceil+2)+m\cdot\lceil (m/\lvert w\rvert)\rceil+m+1$
$CC_{MontExp\_b_{GF(2^m)}}$	$[\lceil (m/\lvert w\rvert)\rceil\cdot(\lvert w\rvert+2)]+(CC_{CopyV2V}-1)+[2\cdot(CC_{MontMult_{GF(2^m)}})]$
$CC_{MontExp\_w_{GF(2^m)}}$	$[\lceil (m/\lvert w\rvert)\rceil]+[\lceil (m/\lvert w\rvert)\rceil\cdot\lvert w\rvert-m+3]+[2\cdot(m-2)\cdot(CC_{MontMult_{GF(2^m)}}-1)]+[(m-2)\cdot(CC_{CopyV2V}-1)]+[(m-3)\cdot(CC_{CopyH2V}-1)]+1$
$CC_{ModAdd\_b_{GF(2^m)}}$	$\lceil (m/\lvert w\rvert)\rceil + 4$
$CC_{ModAdd\_w_{GF(2^m)}}$	$3\cdot\lceil (m/\lvert w\rvert)\rceil + 6$

The prime field $MontMult1$ operation is identical to the $GF(p)$ $MontMult$ operation; therefore the same best and worst case formulas apply.

The computing time formulas of binary field core operations given in clock cycles are listed in Table 6.

The computation time of the $MontMult$ operation in $GF(2^m)$ depends on the specified precision parameter m, the number of active processing units k, and the number of words $e = \lceil (m/|w|)\rceil + 1$ running through the pipeline. Since the additions in $GF(2^m)$ are simple XOR-operations and the most significant bit of the resulting value will never be set after calculation only one formula is given.

While the determination of the Montgomery Parameter r in $GF(2^m)$ differs from the calculation rule for $GF(p)$ it also only depends on the chosen precision m and specified word width $|w|$ parameters.

The $MontR2$ operation in $GF(2^m)$ is based on shifts and possible modular additions whenever the most significant bit of the intermediate value will be set after a shift. The amount of modular additions depends on the Montgomery Parameter r which itself depends on the irreducible polynomial. In order to specify lower and upper computation times a best case and worst case formula is given. The best case assumes that no modular addition operation is required at all, whereas the worst case assumes that a modular addition operation is required after each shift operation.

For the $MontExp$ operation computation time a best case and worst case formula is given. In the best case the exponent

operand is 3 therefore only two Montgomery Multiplications and one $CopyV2V$ operation is necessary. In the worst case the exponent is assumed to be $2^{(m-1)}$ therefore $2 \cdot (m-2) \times MontMult$, $(m-2) \times CopyV2V$ and $(m-3) \times CopyH2V$ operations are required.

For the *ModAdd* operation computation time a best case and absolute worst case formula is given. In the best case the comparator will only have to evaluate one word. In the absolute worst case the comparator will have to evaluate all words to decide that an additional modular addition is necessary. Note that this will only occur if the resulting value after the first addition operation will be identical to the modulus polynomial, which under normal operation conditions will not be the case.

The binary field $MontMult1$ operation is identical to the $GF(2^m)$ $MontMult$ operation; therefore the same formula applies.

6.2. Core Variations. Depending on the needs, in terms of performance, area consumption, supported precisions, and the interfacing structure, different variations of the core can be generated by defining the parameters *MAX_PRECISION_WIDTH*, *WORD_WIDTH* and *MAX_NUM_PUS*. Table 7 lists the resulting number of words e and the possible number of processing units k for the supported prime field precisions $|p|$ and typical word widths $|w|$ of 16, 32 and 64 bit.

In contrast, Table 8 lists the resulting number of words e and the number of possible processing units k for the supported binary field precisions m and typical word widths $|w|$ of 16, 32, and 64 bits. Note that the number of possible processing units for binary fields within the defined core is subjected to a further constraint. Once all bits of A operand have been processed the remaining processing units in the pipeline must be bypassed and the TS and TC words must be directly fed into the CLA adder. Since the result of the CLA adder will be written back to RAM but remaining words must still be read from RAM and fed into the first processing unit, the RAM source and destination signals must never address the same memory location at one time. Therefore the equation $(k - (m \bmod k)) \bmod k$ must hold true to $(k - (m \bmod k)) \equiv 0 \bmod k$, meaning that no processing unit will be bypassed, or $(k - (m \bmod k)) \equiv 1 \bmod k$, meaning that the very last processing unit will be bypassed at the last cycle of A operand bits.

6.3. Core Hardware Footprint. Since all components of the design consist of simple logic elements, the proposed arithmetic core is vendor-neutral. In order to estimate the hardware footprint of different core implementations the design variations have been compiled on Altera and Xilinx FPGAs. Table 9 lists the amount of total logic elements and comprised logic registers for varied values of *WORD_WIDTH* ($|w|$) and *MAX_NUM_PUS* generated for an Altera Cyclone IV (EP4CE115F29C9L) device featuring 114,480 logic elements and 3,981,312 memory bits.

Table 10 lists the amount of total logic elements and comprised logic registers for varied values of *WORD_WIDTH* ($|w|$) and *MAX_NUM_PUS* generated for an Xilinx XC7Z020 (xc7z020clg484-1) device featuring 53,200 logic elements and 106,400 registers. The resulting values demonstrate that the design can compete with other proposed designs, for instance, the one compared in [14, 15]. Furthermore instead of being restricted to only one cryptographic application, the core can handle various algorithms. According to the needs, in terms of area, a suitable solution for a specific implementation can be chosen. The choice will have an impact on power consumption and computing time.

Table 7: Number of words e and amount of possible processing units k depending on precision $|p|$ and word width $|w|$ for $GF(p)$.

GF(p)	**$\|w\|$ = 16**	**$\|w\|$ = 32**	**$\|w\|$ = 64**
$\|p\|$ = 192	$e = 13$ $k = 2, 4, 8$	$e = 7$ $k = 2, 4$	$e = 4$ $k = 2$
$\|p\|$ = 224	$e = 15$ $k = 2, 4, 8$	$e = 8$ $k = 2, 4$	$e = 5$ $k = 2$
$\|p\|$ = 256	$e = 17$ $k = 2, 4, 8$	$e = 9$ $k = 2, 4$	$e = 5$ $k = 2$
$\|p\|$ = 320	$e = 21$ $k = 2, 4,$ $8, 16$	$e = 11$ $k = 2, 4, 8$	$e = 6$ $k = 2, 4$
$\|p\|$ = 384	$e = 25$ $k = 2, 4,$ $8, 16$	$e = 13$ $k = 2, 4, 8$	$e = 7$ $k = 2, 4$
$\|p\|$ = 448	$e = 29$ $k = 2, 4,$ $8, 16$	$e = 15$ $k = 2, 4, 8$	$e = 8$ $k = 2, 4$
$\|p\|$ = 512	$e = 33$ $k = 2, 4,$ $8, 16$	$e = 17$ $k = 2, 4, 8$	$e = 9$ $k = 2, 4$
$\|p\|$ = 768	$e = 49$ $k = 2, 4, 8,$ $16, 32$	$e = 25$ $k = 2, 4,$ $8, 16$	$e = 13$ $k = 2, 4, 8$
$\|p\|$ = 1024	$e = 65$ $k = 2, 4, 8,$ $16, 32$	$e = 33$ $k = 2, 4, 8, 16$	$e = 17$ $k = 2, 4, 8$
$\|p\|$ = 1536	$e = 97$ $k = 2, 4, 8,$ $16, 32, 64$	$e = 49$ $k = 2, 4, 8,$ $16, 32$	$e = 25$ $k = 2, 4,$ $8, 16$
$\|p\|$ = 2048	$e = 129$ $k = 2, 4, 8,$ $16, 32, 64$	$e = 65$ $k = 2, 4, 8,$ $16, 32$	$e = 33$ $k = 2, 4,$ $8, 16$
$\|p\|$ = 3072	$e = 193$ $k = 2, 4, 8,$ $16, 32, 64$	$e = 97$ $k = 2, 4, 8,$ $16, 32, 64$	$e = 49$ $k = 2, 4, 8,$ $16, 32$
$\|p\|$ = 4096	$e = 257$ $k = 2, 4, 8,$ $16, 32, 64$	$e = 129$ $k = 2, 4, 8,$ $16, 32, 64$	$e = 65$ $k = 2, 4, 8,$ $16, 32$

6.4. Core Power Estimation. In order to evaluate the suitability of the proposed core for the application in the IoT area, a power estimation has been conducted using two common frequencies of 100 MHz and 200 MHz for various core variations. Timing analysis yields that the design can reliably be operated with these frequencies. The power consumption characteristics have been derived by applying the PowerPlay Power Analyzer Tool of the Quartus Prime IDE to the

Table 8: Number of words e and amount of possible processing units k depending on precision m and word width $|w|$ for $GF(2^m)$.

$GF(2^m)$	$\|w\| = 16$	$\|w\| = 32$	$\|w\| = 64$
m = 131	$e = 10$ $k = 2, 4$	$e = 6$ $k = 2, 4$	$e = 4$ $k = 2$
m = 163	$e = 12$ $k = 2, 4$	$e = 7$ $k = 2, 4$	$e = 4$ $k = 2$
m = 176	$e = 12$ $k = 2, 4, 8$	$e = 7$ $k = 2, 4$	$e = 4$ $k = 2$
m = 191	$e = 13$ $k = 2, 4, 8$	$e = 7$ $k = 2, 4$	$e = 4$ $k = 2$
m = 193	$e = 14$ $k = 2$	$e = 8$ $k = 2$	$e = 5$ $k = 2$
m = 208	$e = 14$ $k = 2, 4, 8$	$e = 8$ $k = 2, 4$	$e = 5$ $k = 2$
m = 233	$e = 16$ $k = 2$	$e = 9$ $k = 2$	$e = 5$ $k = 2$
m = 239	$e = 16$ $k = 2, 4, 8$	$e = 9$ $k = 2, 4$	$e = 5$ $k = 2$
m = 272	$e = 18$ $k = 2, 4, 8, 16$	$e = 10$ $k = 2, 4, 8$	$e = 6$ $k = 2, 4$
m = 283	$e = 19$ $k = 2, 4$	$e = 10$ $k = 2, 4$	$e = 6$ $k = 2, 4$
m = 304	$e = 20$ $k = 2, 4, 8, 16$	$e = 11$ $k = 2, 4, 8$	$e = 6$ $k = 2, 4$
m = 359	$e = 24$ $k = 2, 4, 8$	$e = 13$ $k = 2, 4, 8$	$e = 7$ $k = 2, 4$
m = 368	$e = 24$ $k = 2, 4, 8, 16$	$e = 13$ $k = 2, 4, 8$	$e = 7$ $k = 2, 4$
m = 409	$e = 27$ $k = 2$	$e = 14$ $k = 2$	$e = 8$ $k = 2$
m = 431	$e = 28$ $k = 2, 4, 8, 16$	$e = 15$ $k = 2, 4, 8$	$e = 8$ $k = 2, 4$
m = 571	$e = 37$ $k = 2, 4$	$e = 19$ $k = 2, 4$	$e = 10$ $k = 2, 4$

final design using default settings of a power toggle rate as well as a power input I/O toggle rate of 12.5%, using a vectorless estimation and a board temperature of 25°C. Table 11 lists the Total Thermal Power Dissipation values for varied *WORD_WIDTH* ($|w|$) and *MAX_NUM_PUS* parameters generated for the Altera Cyclone IV (EP4CE115F29C9L) device. The values are comparable to the ones given in [24] for RSA calculation.

Furthermore it has to be mentioned that the optimization mode in the compiler settings was set to balanced and no specific compiler optimizations regarding power have been turned on. The results show that the core is quite suitable for applications which have special constraints regarding power consumption. According to such needs as well as the desired clock frequency a suitable variation can be implemented. The choice will have an impact on computing time and hardware footprint.

Table 9: Amount of logic elements and logic registers for different core variations (Altera Cyclone IV).

$\|w\|$	MAX_NUM_PUS	Logic Elements	Registers
16	2	3,128	706
16	4	3,523	904
16	8	4,344	1,300
16	16	5,960	2,092
16	32	9,198	3,676
16	64	15,568	6,844
32	2	4,086	988
32	4	4,721	1,314
32	8	5,935	1,966
32	16	8,473	3,270
32	32	13,498	5,878
32	64	23,484	11,094
64	2	6,114	1,557
64	4	7,151	2,139
64	8	9,346	3,303
64	16	13,624	5,631
64	32	22,113	10,287

Table 10: Amount of logic elements and logic registers for different core variations (Xilinx XC7Z020).

$\|w\|$	MAX_NUM_PUS	Logic Elements	Registers
16	2	2,587	755
16	4	2,874	956
16	8	3,467	1,352
16	16	4,623	2,146
16	32	7,208	3,724
16	64	11,934	6,895
32	2	3,367	1,114
32	4	4,014	1,429
32	8	4,694	2,082
32	16	6,645	3,386
32	32	10,177	5,999
32	64	17,770	11,222
64	2	5,098	1,833
64	4	5,869	2,421
64	8	7,585	3,591
64	16	10,654	5,928
64	32	16,871	10,624

6.5. Core Reference Implementation. For the reference implementation a word width of *WORD_WIDTH* = 32 bit was chosen and the maximum number of processing units of the pipeline was set to *MAX_NUM_PUS* = 32. The maximum supported precision width parameter *MAX_PRECISION_WIDTH* was set to 4096 leading to a RAM consisting of 28,672 bits. Table 12 lists the computation time in clock cycles of the reference implementation for RSA application. For RSA public-key operations best case and worst case computation times are given under the assumption that the public exponent is $e = 0x10001$. Therefore during the

Table 11: Total Thermal Power Dissipation (TTPD) values for different core variations (Altera Cyclone IV).

$\|w\|$	MAX_NUM_PUS	TTPD 100 MHz	TTPD 200 MHz
16	2	237.36mW	266.38mW
16	4	239.72mW	297.73mW
16	8	245.97mW	304.40mW
16	16	261.07mW	358.22mW
16	32	289.61mW	389.32mW
16	64	358.76mW	549.60mW
32	2	287.18mW	362.94mW
32	4	294.21mW	375.02mW
32	8	311.36mW	401.65mW
32	16	343.68mW	454.94mW
32	32	398.65mW	568.00mW
32	64	490.90mW	760.71mW

Table 12: Core reference implementation RSA computation times.

	$\|p\| = 2048$	$\|p\| = 3072$	$\|p\| = 4096$
CC_{RSA_pub}	134,128	302,309	529,783
CC_{RSA_priv}	17,925,156	59,094,721	138,504,443
$CC_{CRT-RSA}$	9,194,091	15,667,698	36,131,248

Table 13: Core reference implementation Miller-Rabin computation times.

	$\|p\| = 512$	$\|p\| = 768$	$\|p\| = 1024$
CC_{MR_o1}	1,167,529	1,969,583	4,534,869
CC_{MR_o2}	148	212	276
CC_{MR_i1}	1,246	1,430	2,406
CC_{MR_i2}	148	212	276
	$\|p\| = 1536$	$\|p\| = 2048$	$\|p\| = 3072$
CC_{MR_o1}	7,720,993	17,868,205	58,960,069
CC_{MR_o2}	404	532	788
CC_{MR_i1}	2,790	4,726	10,134
CC_{MR_i2}	404	532	788
	$\|p\| = 4096$		
CC_{MR_o1}	138,267,613		
CC_{MR_o2}	1,044		
CC_{MR_i1}	17,590		
CC_{MR_i2}	1,044		

MontExp operation a total of $17 \times MontMult$, $15 \times CopyH2V$ and $1 \times CopyV2V$ operations will be performed. Since the private exponent is different for varied RSA keys only worst case computation times for the supported precision widths are given. The worst case RSA private key and CRT-accelerated private key computation times assume the worst case clock cycle times of the underlying operations given in previous section.

Table 13 lists the worst computation times in clock cycles of the reference implementation for Miller-Rabin prime testing application for one iteration. Note that the most time consuming operation is part one of the outer loop of Algorithm 2 which will always be performed for each iteration. Depending on the evaluation of the result it might be necessary to execute part two of the outer loop. Furthermore depending on the structure of the prime in question it might be necessary to execute part one and two of the inner loop multiple times.

Table 14: Core reference implementation prime field EC computation times.

	$\|p\| = 192$	$\|p\| = 224$	$\|p\| = 256$
CC_{EC_prep}	5,252	8,281	8,736
$CC_{EC\_mont\_doub}$	742	972	1,234
$CC_{EC\_mont\_add}$	1,468	1,928	2,452
CC_{EC_a2j}	22	24	26
$CC_{EC\_point\_val}$	1,577	2,050	2,587
$CC_{EC\_point\_doub}$	5,613	7,351	9,329
$CC_{EC\_point\_add}$	6,434	8,412	10,662
$CC_{EC\_point\_mult}$	2,288,930	3,499,386	5,077,714
CC_{EC_j2a}	139,233	213,732	311,079
CC_{EC_demont}	734	964	1,226
	$\|p\| = 320$	$\|p\| = 384$	$\|p\| = 512$
CC_{EC_prep}	7,938	10,357	17,575
$CC_{EC\_mont\_doub}$	984	1,364	2,318
$CC_{EC\_mont\_add}$	1,948	2,708	4,612
CC_{EC_a2j}	31	35	44
$CC_{EC\_point\_val}$	2,109	2,895	4,851
$CC_{EC\_point\_doub}$	7,465	10,341	17,533
$CC_{EC\_point\_add}$	8,566	11,842	20,026
$CC_{EC\_point\_mult}$	5,097,858	8,473,906	19,155,090
CC_{EC_j2a}	307,884	514,610	1,172,029
CC_{EC_demont}	974	1,354	2,306

Table 14 lists the computation time in clock cycles of the reference implementation for prime field EC operations for all supported precision widths. The Affine-to-Jacobi Transformation step requires a precision dependent number of clock cycles. For the remaining steps worst case clock cycle times are given. For the Point Multiplication operation an absolute worst case computation time is stated in which a theoretical scalar is hypothesized to be $2^{|p|-1}$, therefore a maximum of $(|p| - 1)$ Point Doubling and $(|p| - 1)$ Point Addition operations would be necessary assuming a simple double and add algorithm.

7. Conclusion and Future Work

A comprehensive adaptable hardware structure for efficient prime finite field and binary finite field arithmetic operations that expand the capabilities of single Montgomery Multiplier hardware designs has been proposed which allows carrying out cryptographic calculations for a large range of different algorithms all based on the same arithmetic unit operations with arbitrary parameters. The approach taken by the proposed core is to combine standard modulo addition / subtraction support with the capability of performing Montgomery Multiplications, full Montgomery Exponentiations,

and the calculation of Montgomery Parameters r and r^2 for arbitrary moduli, bringing together all required arithmetic operations for carrying out a wide range of cryptographic algorithms used today. Through the breakdown of these algorithms individual operation lists have been derived for the arithmetic unit rendering extra precomputations in software unnecessary.

The given values of possible hardware footprint and power consumption for specific core variations allow choosing the proper configuration for a specific implementation. The reference implementation showed that with an internal RAM of merely 3.5 kB the core is capable of performing complete prime field and binary field EC operations for various precision widths of standardised curves. Furthermore the same core configuration is capable of performing (CRT-accelerated) RSA operations for typical precision widths required today, (safe) prime testing/generation, and Diffie-Hellman key exchange operations up to 4096 bit precision widths. The design should further be optimized in terms of power consumption.

However the type of implementation of some core operations, such as the Montgomery Multiplication and especially the Montgomery Exponentiation operation, necessitates additional security considerations, since the calculation times depend on the structure of the processed operands. This makes the design prone to side-channel attacks if security sensitive information, such as private keys, will be processed. But not all operations are critical and must be secured, such as the calculation of the Montgomery Parameters. Therefore during the writing of this article the core will be enhanced to provide a secure calculation bit within the command input word, which, if set, instructs the core to perform the specified arithmetic operation in a time-invariant fashion. In addition, special care has to be taken when defining core operation lists, for instance, for performing elliptic curve Point Multiplication operations. Descriptions performing in a fixed amount of time, e.g., the Montgomery ladder [25], mitigating the risk of timing, and power analysis attacks must be chosen.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] D. Minoli, K. Sohraby, and J. Kouns, "IoT security (IoTSec) considerations, requirements," in *Proceedings of 14th IEEE Annual Consumer Communications Networking Conference (CCNC'17)*, pp. 1006-1007, 2017.

[2] G.-L. Guo, Q. Qian, and R. Zhang, "Different implementations of AES cryptographic algorithm," in *Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 1848–1853, 2015.

[3] M. Bafandehkar, S. M. Yasin, R. Mahmod, and Z. M. Hanapi, "Comparison of ECC and RSA algorithm in resource constrained devices," in *Proceedings of the 2013 3rd International Conference on IT Convergence and Security (ICITCS'13)*, pp. 1–3, 2013.

[4] A. Rupani and G. Sujediya, "A Review of FPGA implementation of Internet of Things," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 4, no. 9, 2016.

[5] B. Halak, S. S. Waizi, and A. Islam, "A survey of hardware implementations of elliptic curve cryptographic systems," Cryptology ePrint Archive 2016/712, 2016.

[6] N. Nedjah and L. de Macedo Mourelle, "A review of modular multiplication methods and respective hardware implementations," *Informatica*, vol. 30, no. 1, pp. 111–129, 2006.

[7] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, 1985.

[8] H. Kaur and C. Madhu, "Montgomery multiplication methods - A review," *Journal of Application or Innovation in Engineeting & Management, IJAIEM*, vol. 2, no. 2, pp. 229–235, 2013.

[9] A. F. Tenca and Ç. K. Koç, "A Scalable Architecture for Montgomery Nultiplication," in *Cryptographic Hardware and Embedded Systems*, vol. 1717 of *Lecture Notes in Computer Science*, pp. 94–108, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[10] E. Savas, A. F. Tenca, and Ç. K. Koç, "A Scalable and Unified Multiplier Architecture for Finite Fields GF(p) and GF(2m)," in *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2000)*, pp. 277–292.

[11] M. Morales-Sandoval and A. D. Perez, "Novel algorithms and hardware architectures for Montgomery Multiplication over GF(p)," Cryptology ePrint Archive 2015/696, 2015.

[12] R. Cramer, *Public Key Cryptography – PKC 2008*, vol. 4939, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[13] A. Mrabet, "A Systolic Hardware Architectures of Montgomery Modular Multiplication for Public Key Cryptosystems, Cryptology ePrint Archive," *Report 2016/487*, 2016.

[14] Z. Liu et al., "A tiny RSA coprocessor based on optimized systolic Montgomery architecture," in *Proceedings of the International Conference on Security and Cryptography (SECRYPT'11)*, 2011.

[15] G. D. Sutter, J.-P. Deschamps, and J. L. Imana, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 7, pp. 3101–3109, 2011.

[16] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[17] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128–138, 1980.

[18] J. von zur Gathen and I. E. Shparlinski, "Generating safe primes," *Journal of Mathematical Cryptology*, vol. 7, no. 4, pp. 333–365, 2013.

[19] W. Diffie, W. Diffie, and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[20] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[21] BSI - Technical Guidline, "Cryptographic Mechanisms: Recommendations and Key Lengths," BSI TR-02102-1, 2017.

[22] D. Wulansari, M. A. Muslim, and E. Sugiharti, "Implementation of RSA algorithm with chinese remainder theorem for modulus n 1024 bit and 4096 bit," *International Journal of Computer Science and Security*, vol. 10, no. 5, pp. 186–194, 2016.

[23] V. S. Miller, "Use of elliptic curves in cryptography," in *Proceedings of the Conference on the Theory and Application of Cryptographic Techniques, CRYPTO 1985*, pp. 417–426, 1985.

[24] B. Zhou, M. Egele, and A. Joshi, "High-performance low-energy implementation of cryptographic algorithms on a programmable SoC for IoT devices," in *Proceedings of the 2017 IEEE High-Performance Extreme Computing Conference (HPEC)*, 2017.

[25] M. Joye and S. Yen, "The Montgomery Powering Ladder," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, pp. 291–302, 2002.